J. Symbolic Computation (1993) **15**, 451–462

An Efficient Algorithm and Parallel Implementations for Binary and Residue Number Systems

C.N. ZHANG* , B. SHIRAZI** AND D.Y.Y. YUN***

**Dept. of Computer Science, University of Regina, Regina Sask., Canada*

***Dept. of Computer Science and Engineering,*
University of Texas at Arlington, Arlington TX, USA

****Dept. of Electrical Engineering, University of Hawaii at Manoa, Hololulu Hawaii, USA*

(Received 21 January 1992)

Arithmetic units based on a Residue Number System (RNS) are fast and simple, and therefore attractive for use in digital signal processing and symbolic computation applications. However, RNS suffers from overheads of converting numbers to and from residue system. We present a new simple and uniform computation formula for both conversion from RNS to binary and vice versa. Two levels of parallelism for VLSI hardware design of the proposed algorithm are introduced.

1. Introduction

Residue Number Systems (RNS) provide an alternative method for arithmetic on large integers. The advantage of residue representation is that addition, subtraction, and multiplication can be performed in an exact modular fashion without carry propagation. The RNS approach to large number computation involves a set of parallel independent computations which can result in a significant speedup relative to the use of conventional algorithms [1-4]. The use of RNS is interesting to consider for symbolic mathematical computation where computation time is often dominated by that required to multiply large integers. In the RNS approach, initial integer data is converted to the RNS representation where it remains throughout computation. Values need only be converted back to binary representation for the communicating results to the user or when an alternate representation is required. Major disadvantages of the RNS is that division cannot be performed, and that overflow detection and sign detection are difficult. In addition, RNS requires the overhead of converting binary representation to RNS and vice versa. Yun [5] showed that these two conversions can be viewed as a pair of mapping and corresponding inverse mapping. Both have the same lower bound of time complexity, $O(n \log n)$, where n is the number of moduli in a RNS. To limit this overhead, several techniques for fast implementations of the conversions have been proposed. Taylor [6] has assumed that dynamic range of RNS is $M = 2^a - 2^b$ with a and b being integers. Because

0747–7171/93/040451 + 12 $08.00/0

of this particular value of M, the overflow detection and correction operations can be implemented by a Programmable Logic Array (PLA). However, this design is still expensive and the restrictions imposed on M limit the selection of the system moduli. Vu [7] has expressed the modular summation in terms of quotients and remainders with respect to a properly chosen divisor. Instead of the sum modular M, his method computes two binary sums corresponding to remainders and quotients. The final recombination step requires another table look-up and some binary additions. Vu's algorithm assumes that one of the moduli of the residue system has to be a power of two. In this paper we show that the conversion from the binary representation to RNS and vice versa can be transformed into a common simple formula by using VLSI techniques. In addition, we show that the parallelism for the proposed algorithm can be implemented at two different levels. At the architecture level, a linear systolic array and a binary tree systolic array implementations are proposed. At the computation level, carry-save addition is used to replace binary addition to achieve better performance.

2. Conversion from RNS to Binary System

A RNS consists of n relative prime moduli $\{m_1, m_2, \ldots m_n\}$. Any integer S, $0 \leq S \leq M-1, (M = \prod_{i=1}^{n} m_i)$ is represented in this system by an n-tuple $(x_1, x_2, \ldots, x_n)$ where $x_i = S \bmod m_i$, $i = 1, 2, \ldots, n$. The total number M of integers that can be unambiguously encoded is called the dynamic range of RNS. The conversion of an integer S from its residue number representation $(x_1 \ldots x_n)$ into ordinary binary number using the Chinese Remainder Theorem (CRT) can be expressed as follows: $S = \sum_{i=1}^{n} \frac{M}{m_i} (\omega_i x_i \bmod m_i) \bmod M$, where ω_i is a constant corresponding to m_i such that $\left(\frac{M}{m_i}\right) \omega_i \equiv 1 \ (mod\ m_i)$. The notation $a \equiv b \ (mod\ M)$ means congruence (that is, $|a - b| = cM$ where $c \geq 0$ is an integer). Since m_i is usually small, the value of $t_i = \frac{M}{m_i} (\omega_i x_i \bmod m_i)$ can be viewed as a function of variable x_i which can be evaluated by accessing a precomputed look-up table (ROM) with a small address space. Symbolic algebra applications often need to deal with large integers, say 1000 bits of binary numbers. For 1000 bit integers, we can choose $2^{20} \leq m_i < 2^{21}$ for all moduli. A total of 50 moduli would be sufficient. To produce t_i, the look-up table can be implemented by cascading 5 smaller ROMs, each with 20 inputs and 200 outputs (Fig. 1). Thus, the conversion based on CRT becomes a problem of computing sums of n integers modulo M, $S = \sum_{i=1}^{n} t_i \bmod M$, where $0 \leq t_i < M$ (Fig. 2).

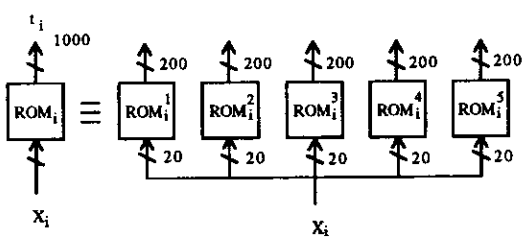


Fig. 1. An implementation of look-up table with 1000 outputs by five ROM's each of which with 200 outputs.

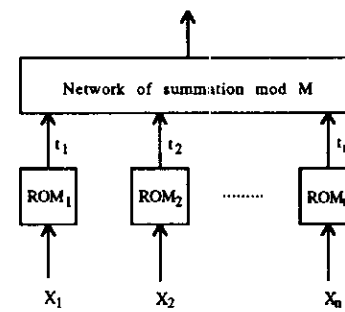


Fig. 2. A simplified residue conversion based on CRT.

The integer M can be represented as $M = 2^k - p$, where k is the smallest integer such that $M \leq 2^k$, and p is constant with respect to M. Since $2^{k-1} < M < 2^k$, we have $0 < p < 2^{k-1}$.

First, let us consider different ways to calculate modular addition ($A + B \ mod \ M$). The straightforward method is as follows:

Method 1: (1) $C := A + B$; (2) $D := C - M$; (3) **if** $D \geq 0$ **then return** D **else return** C.

This method requires one binary adder, used twice per modular addition. Computing $\sum_{i=1}^{n} t_i \ mod \ M$ requires $2n \ t_b$ units of time, where t_b is the time delay for one binary addition.

The next method is a modification of method 1 which performs Step 1 and Step 2 in parallel using two binary adders:

Method 2: (1) $C := A + B$ and $D := A + B - M$; (2) **if** $D \geq 0$ **then return** D **else return** C.

Computing $\sum_{i=1}^{n} t_i \ mod \ M$ requires $n \ t_b$ units of time using two adders. The following method is based on the concept of biased addition which makes the detection of overflow (i.e. detecting $D \geq 0$ condition) easier. Subtraction is still required if there is no overflow:

Method 3: (1) $\tilde{C} := A + B + p$; (2) **if** $\tilde{C} \geq 2^k$ **then return** $\tilde{C} - 2^k$ **else return** $\tilde{C} - p$.

Method 3 requires one binary adder. The average time delay per operation is 1.5 times binary additions, since the sum of A and B may or may not be larger than M. Therefore, computing $\sum_{i=1}^{n} t_i \ mod \ M$ requires 1.5 n t_b units of time on average. In the following, we propose an algorithm which computes $\sum_{i=0}^{n} t_i \ mod \ M$ in $n \ t_b$ time delay using one binary adder.

Define partial sum S_i such that $S_0 = 0$ and $S_i = (S_{i-1} + t_i) \ mod \ M$ for $1 \leq i \leq n$. Define biased partial sum $\tilde{S}_i$ such that $\tilde{S}_i = S_{i-1} + t_i + p$ for all $1 \leq i \leq n$. For any $1 \leq i \leq n$, it follows that [8] $p \leq \tilde{S}_i < 2^{k+1} - p$. Also, S_i and $\tilde{S}_i$ have the following relationship: $S_i = \tilde{S}_i - 2^k$ if $\tilde{S}_i \geq 2^k$ and $\tilde{S}_i - p$ if $\tilde{S}_i < 2^k$. Thus, we have the following iterative formula.

Theorem 1. $\tilde{S}_1 = t_1 + p$ and for all $i \geq 2$,

$$\tilde{S}_i = \begin{cases} (\tilde{S}_{i-1} - 2^k) + t_i + p & if \ \tilde{S}_{i-1} \geq 2^k \\ \tilde{S}_{i-1} + t_i & if \ \tilde{S}_{i-1} < 2^k \end{cases}$$

Note that the conditions $\tilde{S}_i \geq 2^k$ and $\tilde{S}_i < 2^k$ can be easily determined by checking the overflow condition (k−th bit) of the biased partial sum $\tilde{S}_i$ (since $\tilde{S}_i$ is in the range of p and $2^{k+1} - p - 1$). In practice these two conditions can be respectively represented by a flag $D = 1$ and $D = 0$. Also note that the operation of $\tilde{S}_i - 2^k$ simply drops the k−th bit of $\tilde{S}_i$. Formally, a new algorithm for computing $S = \sum_{i=1}^{n} t_i \ mod \ M$ is as follows.

Algorithm 1.

Step 1 [initialization]: $D := 1, \ S := 2^k$;
Step 2 [iterations]:
for $i = 1$ **to** n **do** {
2.1 **if** $D = 1$ **then** $S := (S - 2^k) + t_i + p$

else $S := S + t_i$;
2.2 **if** $S \geq 2^k$ **then** $D := 1$ **else** $D := 0$;
}
Step 3 [final correction]: **if** $D = 1$ **then** $S := S - 2^k$ **else** $S := S - p$.

This algorithm is carried out in n steps, each step performing a binary addition with either two or three operands. The addition of three operands can be implemented by cascading a carry-save adder and conventional binary adder [8].

3. Hardware Implementations

In this section, we present two hardware implementations for the proposed algorithm suitable for VLSI techniques.

3.1. A Linear Systolic Array Implementation

A systolic array is a computing network composed of processing elements (PEs) and local inter-connections between them. It exploits computation concurrencies, including multiple processing and pipeline processing, and is therefore suitable for the computation intensive problems. If we directly implement Algorithm 1 by a linear systolic array, then the systolic array requires two different types of PEs corresponding to two different steps (an iteration step and a final correction step) in Algorithm 1. To simplify the systolic array implementation, we modify Algorithm 1 as follows. Let $t_{n+1} = -p$. Step 3 of Algorithm 1 can be rewritten as: **if** $D = 1$ **then** $S := (S - 2^k) + t_{n+1} + p$ **else** $S := S + t_{n+1}$ (the same formula as Step 2.1 of Algorithm 1), thus removing Step 3 of Algorithm 1:

Algorithm 1a.

Step 1 [initialization]: $D := 1$, $S := 2^k$, $t_{n+1} = -p$;
Step 2 [iterations]:
for $i = 1$ **to** $n+1$ **do** {
2.1 **if** $D = 1$ **then** $S := (S - 2^k) + t_i + p$
else $S := S + t_i$;
2.2 **if** $S \geq 2^k$ **then** $D := 1$ **else** $D := 0$;
}.

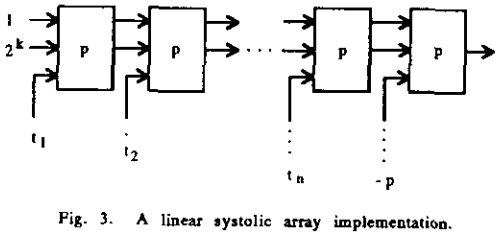


Fig. 3. A linear systolic array implementation.

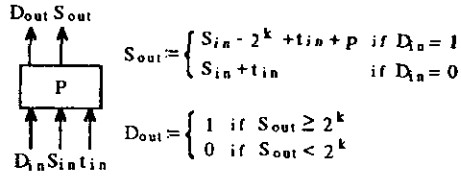


Fig.4. The logic diagram of PE in the systolic array in Fig.3.

A linear systolic array implementation of Algorithm 1a is shown in Fig. 3. The block diagram of PE of the systolic array is shown in Fig. 4 which computes the iteration step of Algorithm 1a in one clock cycle. The input control signal D_{in} indicates an overflow produced in the previous computation, and D_{out} is a control output which represents an overflow ($S_{out} \geq 2^k$) produced during the current computation. In general, this systolic array requires $n + 1$ PEs. The time latency of the systolic array defined as the number of clock cycles required between the first input data entering the array and the first result leaving from that array, is $n + 1$.

3.2. A Binary Tree Systolic Array Implementation

Next, we present a binary tree systolic array implementation of Algorithm 1 that speeds up the computation time (time latency) from linear to logarithmic. Here, we assume $n = 2^m$.

Define the partial sum from the j_1-th input to the j_2-th input as: $S(j_1, j_2) = \sum_{i=j_1}^{j_2} t_i \ mod\ M$, where $j_2 \geq j_1$. Since n is a power of 2, one can always compute the partial sums in parallel level-by-level according to a binary tree structure. For example, with $n = 8$, the first level computes S(1,2), S(3,4), and S(5,6) and S(7,8) in parallel. Level two computes S(1,4) and S(5,8) in parallel and S (1,8) is computed in the final level. The partial sum can be defined as: $S(j_1, j_2) = S(j_1, w) + S(w+1, j_2) \ mod\ M$ where $w = \frac{j_2 - j_1 - 1}{2} + j_1$, $j_2 \geq j_1$ and $j_2 - j_1 - 1$ is even. Similarly, the biased partial sum from j_1-th input to the j_2-th input is defined as: $\tilde{S}(j_1, j_2) = S(j_1, w) + S(w+1, j_2) + p$. It can be easily shown that $p \leq \tilde{S}(j_1, j_2) < 2^{k+1} - p$ and that $S(j_1, j_2) = \tilde{S}(j_1, j_2) - 2^k$ *if* $\tilde{S}(j_1, j_2) \geq 2^k$ and $\tilde{S}(j_1, j_2) - p$ *if* $\tilde{S}(j_1, j_2) < 2^k$. Thus, we have:

Theorem 2.

$$\tilde{S}(j_1, j_2) = \begin{cases} \tilde{S}(j_1, w) + \tilde{S}(w+1, j_2) - p & \text{if case 1} \\ (\tilde{S}(j_1, w) - 2^k) + \tilde{S}(w+1, j_2) & \text{if case 2} \\ \tilde{S}(j_1, w) + (\tilde{S}(w+1, j_2) - 2^k) & \text{if case 3} \\ (\tilde{S}(j_1, w) - 2^k) + (\tilde{S}(w+1, j_2) - 2^k) + p & \text{if case 4} \end{cases}$$

where $w = \frac{j_2 - j_1 - 1}{2} + j_1$, $j_2 > j_1$, *case* 1 = $\tilde{S}(j_1, w) < 2^k$ and $\tilde{S}(w+1, j_2) < 2^k$, *case* 2 = $\tilde{S}(j_1, w) \geq 2^k$ and $\tilde{S}(w+1, j_2) < 2^k$, *case* 3 = $\tilde{S}(j_1, w) < 2^k$ and $\tilde{S}(w+1, j_2) \geq 2^k$, and *case* 4 = $\tilde{S}(j_1, w) \geq 2^k$ and $\tilde{S}(w+1, j_2) \geq 2^k$.

Finally, we have: $S = S(1, n) = \tilde{S}(1, n) - 2^k$ if $\tilde{S}(1, n) \geq 2^k$; $\tilde{S}(1, n) - p$ if $\tilde{S}(1, n) < 2^k$. Formally, the computation $S = \sum_{i=1}^{n} t_i \ mod\ M$ can be described by following algorithm.

Algorithm 1b (a parallel version of Algorithm 1)

Step 1. [initialization]: **for** i := 1 **to** m **do** $S_i := t_i + 2^k$;
Step 2 [iteration]:

```
for i := 1 to m do {
  for j := 1 step 2^i to n-2i+1 do {
    case of
      (i) S_j ≥ 2^k and S_{j+2^{i-1}} ≥ 2^k :  S_j := S_j - 2^k + S_{j+2^{i-1}} - 2^k + p
      (ii) S_j ≥ 2^k and S_{j+2^{i-1}} < 2^k :  S_j := S_j - 2^k + S_{j+2^{i-1}}
      (iii) S_j < 2^k and S_{j+2^{i-1}} ≥ 2^k :  S_j := S_j + S_{j+2^{i-1}} - 2^k
```

(iv) $S_j < 2^k$ *and* $S_{j+2^{i-1}} < 2^k$: $S_j := S_j + S_{j+2^{i-1}} - p$

} }

Step 3 [final correction] **if** $S_1 \geq 2^k$ **then** $S := S_1 - 2^k$ **else** $S := S_1 - p$;

Similarly, Step 3 of Algorithm 1b can be removed by defining $S_{n+1} = 2^k - p$ and adding one more iteration step. Fig. 5 shows an example of a binary tree systolic array for case $n = 8$. The organization has a complete binary tree structure consisting of n-1 uniform PEs, with an additional PE above the root node of the tree. The block diagram of the PE is shown in Fig. 6 which performs the iterative formula according to the the combination of the two control signals, α_{inL} and α_{inR}. Note that the first row of array of Fig. 5 is used to compute the biased partial sums, $\tilde{S}(1, 2)$, $\tilde{S}(3, 4)$, $\tilde{S}(5, 6)$ *and* $\tilde{S}(7, 8)$. According to the function of the PE, those biased partial sums can be performed by *setting both of input control signals* α_{inR} *and* α_{inL} *to* 1, *and adding the most significant* bit (2^k) to each t_i. For example, $\tilde{S}(1, 2) = (t_1 + 2^k) - 2^k + (t_2 + 2^k) - 2^k + p$.

In general, the binary tree systolic array implementation requires n PEs, and the time latency is log n + 1.

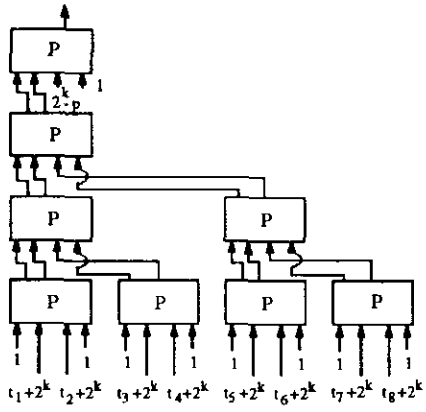


Fig.5. A binary tree systolic array implementation.

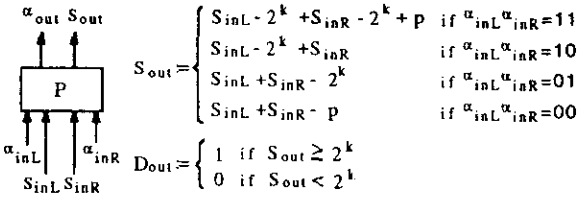


Fig.6. The logic diagram of PE in the systolic array in Fig.5.

4. Replacing Binary Addition by Carry-Save Addition

Now we consider how to introduce parallelism in the proposed algorithm at the computation level. The above proposed algorithm and its versions assume that binary addition and subtraction are primitive operations. The use of a redundant number representation is advantageous in order to avoid carry propagations. One such redundant model is the binary carry-save representation. In this form, a k+1-bit number S is represented as a pair of two k-bit integers: (S^p, S^c) such that $S = S^c + S^p$, where S^c is the carry value, and S^p represents the place (local) summation. To add a binary number $B = \sum_{i=0}^{k-1} b_i$ *to an integer in carry-save form* (S^p, S^c), $(S^p, S^c) = S^p + S^c + B$, *one computes:* $s_i^p = s_i^p \oplus s_i^c \oplus b_i$ and $s_{i+1}^c = s_i^p s_i^c + s_i^p b_i + s_i^c b_i$, for all $0 \leq i \leq k-1$, where $\oplus$ and $+$ are logic X-OR and logic OR, respectively. If B itself is in carry-save representation, $B = (B^p, B^c)$, the summation can be computed in two levels of bit-wise additions by two cascaded 3 to 2 carry save adders (CAS1 and CAS2) without carry propagation beyond two positions. For the representation of negative numbers, as well as in subtraction, a complement form can be used.

Recall that during each iteration of the proposed algorithm, it is necessary to detect

the overflow of the intermediate value $\tilde{S}_i$. To determine the condition $\tilde{S}_i \geq 2^k$, one must find the actual value $\tilde{S}_i$ by adding the two integers together. Again, binary addition is required in each iteration. Note that in the derivation of Algorithm 1, the partial sum S_i is defined in the range between zero and $M-1$. At the same time the range of the corresponding biased partial sum $\tilde{S}_i$ is limited from p to $2^{k+1}-p-1$. Since the problem is dealing with modulo arithmetic, range constraints are not necessary. Instead, one can allow the partial sum and biased partial sum to be in a larger range during the iterations, and correct the result in the final step such that the detection of the overflow during the iterations becomes trivial. The details of detection of overflow will become clear after introducing a modified algorithm below.

Define the partial sum in carry-save form (S_i^c, S_i^p) as: $(S_i^p, S_i^c) \equiv \sum_{j=1}^{i} t_j \ (mod\ M)$ where $0 \leq S_i^c < 2^k$, $0 \leq S_i^p < 2^k$. Note that due to redundant representation, the partial sum in carry-save form defined above is not unique. Any pair of two k-bit integers, (S_i^P, S_i^c) which satisfies this definition is a valid one. Define the biased partial sum of carry-save form $(\tilde{S}_i^p, \tilde{S}_i^c)$ as: $(\tilde{S}_i^p, \tilde{S}_i^c) = S_{i-1}^p + S_{i-1}^c + t_i + p$. Under Definitions of partial sum and biased partial sum in carry-save form, it follows that $\tilde{S}_i^p < 2^k$, $p < \tilde{S}_i^c + \tilde{S}_i^p < 3 \times 2^k$, and if $2^{k+1} \leq \tilde{S}_i^c$ then $\tilde{S}_i^p < 2^{k-1}$ [8]. Also, it follows that: $(S_i^p, S_i^c) = \tilde{S}_i^c - p + \tilde{S}_i^p$ if $\tilde{S}_i^c < 2^k$; $\tilde{S}_i^c - 2^k + \tilde{S}_i^p$ if $2^k \leq \tilde{S}_i^c < 2^{k+1}$; $\tilde{S}_i^c - 2^{k+1} + \tilde{S}_i^p + p$ if $2^{k+1} \leq \tilde{S}_i^c$. Combining these results one can derive the following iterative formula:

Theorem 3. $(\tilde{S}_1^p, \tilde{S}_1^c) = t_1 + p$ and for all $i \geq 2$

$$(\tilde{S}_i^p, \tilde{S}_i^c) = \begin{cases} \tilde{S}_{i-1}^c + \tilde{S}_{i-1}^p + t_i & if\ \tilde{S}_i^c < 2^k \\ (\tilde{S}_{i-1}^c - 2^k) + \tilde{S}_{i-1}^p + t_i + p & if\ 2^k \leq \tilde{S}_i^c < 2^{k+1} \\ (\tilde{S}_{i-1}^c - 2^{k+1}) + \tilde{S}_{i-1}^p + t_i + 2p & if\ 2^{k+1} \leq \tilde{S}_i^c \end{cases}$$

The overflows during the iteration steps can be represented by two flags D_1 and D_2 which are set or reset according to whether or not overflow occurs from the the most significant bit of CAS1 and CAS2 shown in Fig. 7.

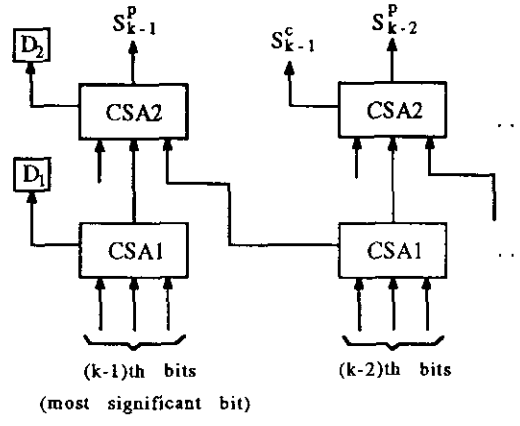


Fig.7. Overflow detection of the four-to-tow carry-save adder.

Since $S_n^c + S_n^p$ may be greater than M, additional correction steps are required to ensure that the final result is in the range of zero and $M - 1$. The final correction operations are explained in the following five steps. The addition operation specified in these steps is the binary addition and not carry-save addition:

Step1.

$$\tilde{S}_n = \begin{cases} \tilde{S}_n^c + \tilde{S}_n^p & \text{if case } a \\ \tilde{S}_n^c + \tilde{S}_n^p - 2^k + p & \text{if case } b \\ \tilde{S}_n^c + \tilde{S}_n^p - 2^{k+1} + 2p & \text{if case } c \quad \text{where} \end{cases}$$

case $a = (\tilde{S}_n^c < 2^k)$ and $(\tilde{S}_n^c < 2^{k-1}$ *or* $\tilde{S}_n^p < 2^{k-1})$, *case* $b = (2^k \leq \tilde{S}_n^c < 2^k + 2^{k-1}$ and $\tilde{S}_n^p < 2^{k-1})$ or $(2^{k-1} \leq \tilde{S}_n^c < 2^k$ *and* $2^{k-1} \leq \tilde{S}_n^p < 2^k)$, *case* $c = 2^{k+1} \leq \tilde{S}_n^c$.

Step 2.

$$\tilde{S}^* = \begin{cases} \tilde{S}_n - 2^k + p & \text{if } \tilde{S}_n \geq 2^k \\ \tilde{S}_n & \text{if } \tilde{S}_n < 2^k \end{cases}$$

Step 3.

$$S = \begin{cases} \tilde{S}^* - 2^k & \text{if } \tilde{S}^* \geq 2^k \\ \tilde{S}^* - p & \text{if } \tilde{S}^* < 2^k \end{cases}$$

Finally, the modified algorithm using carry-save form is as follows:

Algorithm 1c (a version of Algorithm 1 using the carry-save representation)

Step 1. [initialization]

$\{S^p := p,\ S^c := 0,\ P := p,\ 2P := 2p,\ T := t_1,\ D_1 := 0,\ D_2 := 0,\ K := 2^{n-1}\}$;

Step 2. [iterations] (by a 4 to 2 carry-save adder)

while $K \neq 0$ **do** {

$$(S^p, S^c) := \begin{cases} S^p + S^c + T & \text{if } \bar{D}_2 \bar{D}_1 \\ S^p + (S^c - 2^k) + T + P & \text{if } D_2 \oplus D_1 \\ S^p + (S^c - 2^{k+1}) + T + 2P & \text{if } D_2 D_1 \ ; \end{cases}$$

set or reset D_1 and D_2 (according to the overflows of the CSA's) ;

right-shift K ; T **accepts** next input ;

}

Step 3. [final correction] as previously defined.

5. Conversion from Binary Representation into RNS Representation

We now address the conversion from the binary representation into residue number representation. Let system moduli be $\{m_1, m_2, \ldots m_n\}$ where $2^{l_i-1} \leq m_i < 2^{l_i}$, $1 \leq i \leq n$. $2^{k-1} < M < 2^k$, and $M = \prod_{i=1}^{m} m_i$. The problem of converting an integer B from the binary representation into the residue number representation can be described as follows: **given** integer $B = \sum_{i=0}^{k-1} b_i 2^i$, $b_i \in \{0, 1\}$, **compute** the residue number representation of B: $(x_1, x_2, \ldots x_n)$ where $x_i = B \bmod m_i$, $1 \leq i \leq n$. The following approach is based on the representation of the power of two modulo m_i. By definition, $x_i = B \bmod m_i = \sum_{i=0}^{k-1} 2^i b_i \bmod m_i$. Since $m_i < 2^{l_i}$, m_i can be represented by l_i bits. Let $\sum_{j=0}^{l_i-1} 2^i b_j = r_0^{(i)}$ and $2^{l_i+j-1} \bmod m_i = r_j^{(i)}$, $1 \leq j \leq k - l_i$. We

have $x_i = (r_0^{(i)} + r_1^{(i)} b_{l_i} + \ldots + r_{k-l_i}^{(i)} b_{k-1}) \bmod m_i = \sum_{j=0}^{k-l_i} t_j^{(i)} \bmod m_i$, where $t_0^{(i)} = r_0^{(i)}$, and for $1 \leq j \leq k - l_i$: $t_j^{(i)} = 0$ *if* $b_{l_i+j-1} = 0$ *and* $r_j^{(i)}$ otherwise. Each $r_j^{(i)}$ can be precomputed and stored in a register R_j, $j = 1, 2, \ldots, k - l_i$. The problem of converting from the binary representation into residue representation can thus be transformed to the problem of modular summation of $k - l_i + 1$ integers as shown in Fig. 8. This computational format is exactly the same as the one explained in conversion from RNS to binary representation. The proposed algorithm and its parallel designs therefore can be applied here as well.

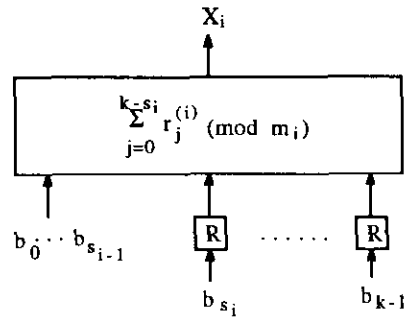


Fig.8. A simplified conversion from residue into binary representation.

6. Extendable Architectures

In most RNS applications, the dynamic range $\prod_{j=1}^{n} m_j$ of a RNS is predetermined by the set of moduli selected. In certain circumstance, however, it is desirable to use two or more hardware units each of which was designed for conversions of a RNS with fixed dynamic range, to calculate conversions for some RNS with larger dynamic range. We now explain that the proposed architectures can be extended to compute conversions of residue number systems in which a larger dynamic range is required. For simplicity of explanation, we use the linear systolic array shown in Fig. 3 to illustrate the idea. The same idea can be applied to the rest of proposed architectures. Assume that a linear systolic array as shown in Fig. 3 is designed for a RNS with dynamic range $2^{k-1} < M < 2^k$ where $M = \prod_{i=1}^{n} m_i$. Fig.9 shows corresponding logic circuit design of the PE whose logic diagram is shown in Fig. 4. A k-bit register (latch) and a one bit flag (flip-flop) are needed for the purpose of pipelining synchronization. C_{in} and C_{out} are initial carry and end carry (overflow) of the k-bit binary adder. In the previous design (for a RNS with fixed size dynamic range), the C_{in} was connected to zero and C_{out} was ignored.

Suppose that there is another RNS with a set of moduli $(m'_1, m'_2, \ldots, m'_h)$ and dynamic range $2^{2k-1} < M' = \prod_{i=1}^{h} m'_i < 2^{2k}$. Let $M' = 2^{2k} - p'$ where $0 < p' < 2^{2k-1}$. According to the proposed approach, a number S' in this RNS representation can be converted into its binary representation by computing $S' = \sum_{i=1}^{h} t'_i$ where t' is a output of a ROM (If the size of the ROM is too large, it can be implemented by several small size ROM as shown in Fig. 1) which can be implemented by a linear systolic array similar to that in Fig. 3 except with h + 1 PEs. The logic diagram of the new PE is similar to that shown in Fig. 4, except that the input and output data (S'_{in}, t'_{in} *and* S'_{out}) are 2k bits long. Let $S'_{in} = S'^{H}_{in} 2^k + S'^{L}_{in}$, $S'_{out} = S'^{H}_{out} 2^k + S'^{L}_{out}$, $t'_{in} = t'^{H}_{in} 2^k + t'^{L}_{in}$,

and $p' = p^H 2^k + p^L$ where all of S_{in}^H, S_{in}^L, S_{out}^H, S_{out}^L, p^H, p^L, t_{in}^H and t_{in}^L are k-bit integers. The function of the new PEs can be rewritten as:

$$C_{in}^L := 0 \qquad C_{in}^H := C_{out}^L$$

$$S_{out}^L := \begin{cases} S_{in}^L + t_{in}^L + p^L & if\ D_{in}' = 1 \\ S_{in}^L + t_{in}^L & if\ D_{in}' = 0 \end{cases} \quad and \quad S_{out}^H := \begin{cases} S_{in}^H + t_{in}^H + p^H & if\ D_{in}' = 1 \\ S_{in}^H + t_{in}^H & if\ D_{in}' = 0 \end{cases}$$

$$C_{out}^L := \begin{cases} 1 & if\ S_{out}^L \geq 2^k \\ 0 & if\ S_{out}^L < 2^k \end{cases} \qquad D_{out}^L := \begin{cases} 1 & if\ S_{out}^H \geq 2^k \\ 0 & if\ S_{out}^H < 2^k \end{cases}$$

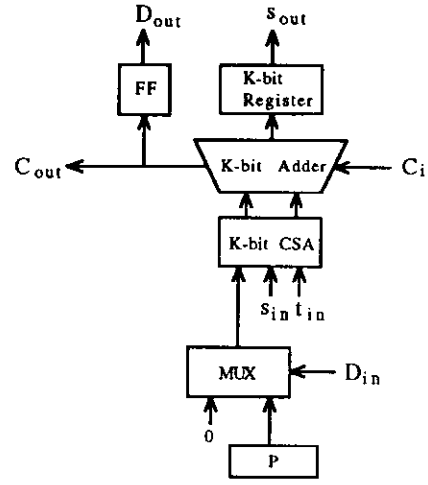
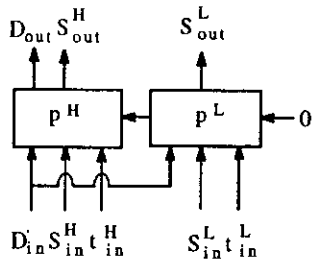


Fig. 9. Logic circuit design of PE in Fig.4.

Thus the new PEs can be constructed by connecting two PEs designed for k-bit operands as shown in Fig. 10. Therefore, the systolic array for the RNS with h moduli and 2k-bit dynamic range can be implemented by a total of 2 (h + 1) PEs.



$$C_{in}^L := 0 \qquad C_{in}^H := C_{out}^L$$

$$S_{out}^H := \begin{cases} S_{in}^H + t_{in}^H + p^H & \text{if } D_{in}'=1 \\ S_{in}^H + t_{in}^H & \text{if } D_{in}'=0 \end{cases} \quad \text{and} \quad S_{out}^L := \begin{cases} S_{in}^L + t_{in}^L + p^L & \text{if } D_{in}'=1 \\ S_{in}^L + t_{in}^L & \text{if } D_{in}'=0 \end{cases}$$

$$C_{out}^L := \begin{cases} 1 & \text{if } S_{out}^L \geq 2^k \\ 0 & \text{if } S_{out}^L < 2^k \end{cases} \qquad D_{out}' := \begin{cases} 1 & \text{if } S_{out}^H \geq 2^k \\ 0 & \text{if } S_{out}^H < 2^k \end{cases}$$

Fig.10. The new PE constructed by two PEs shown in Fig.9.

Table 1. Summary of proposed architectures.

	linear array (binary adder)	tree array (binary adder)	linear array (CSA)	tree array (CSA)
time delay of PE ($M < 2^k$)	t_b	t_b	t_c	t_c
computation time (with n moduli)	$t_b\ (n + 1)$	$t_b\ (logn + 1)$	$O(\ t_c\ n\ )$	$O(\ t_c\ log\ n\ )$
number of PEs (with n moduli)	n + 1	n	O (n)	O (n)
complexity of PE	very simple	simple	complex	more complex
types of PE	one	one	more than one	more than one
extendable	yes	yes	yes	yes
time delay of PE ($M < 2^{2k}$)	$2\ t_b$	$2\ t_b$	t_c	t_c

Note that since a 2k-bit addition is done by two cascaded k-bit adders, the second adder has to wait until that the end carry (C_{out}^L) produced by the first adder is available. As a consequence, the time delay of the new PE is $2\ t_b$ where t_b is the time delay of the binary adder with k-bit operands. The technique of replacing binary additions by carry-save additions described in section 4 could eliminate this overhead. However, since binary additions are still required in Step 3 of Algorithm 3c, each has to be implemented by a pipelining architecture. Thus, more number of PEs and more types of PEs are required. A summary of the proposed architectures is given in Table 1.

where t_b and t_c are the time delays of binary adder and carry save adder respectively.

7. Conclusion

The two conversions between the binary representation and the residue number representation can be transformed into the same simple formula, namely modular summation of some number of integers, supported by VLSI technique. We presented an efficient algorithm to compute this formula by totally eliminating the modular operations. To optimize the performance of this algorithm, parallel implementations have been considered at two different levels. At the architecture level, a linear systolic array and a binary tree systolic array are developed. At the computation level, by extending the concept of the binary biased addition to the biased carry-save addition, the binary addition which is the basic operation of the proposed algorithm can be replaced by the carry-save addition. We show that all proposed hardware implementations have extendable capability.

References

[1] Hwang, K. (1979). *Computer Arithmetic-Principles, Architecture, and Design*, John Wiley.

[2] Jenkins, W.K. (1973). A Technique for the Efficient Generation of Projections for Error Correcting Residue Codes. *IEEE Trans. Computers*, Vol C-22, pp762-767.

[3] Szabo, N.S. and R. I. Tanaka (1967). *Residue Arithmetic and its Applications to Computer Technology*. McGraw-hill, New York.

[4] Tseng, B., G.A. Julin and W.C. Miller. (1979). Implementation of FFT Structures Using the Residue Number System. *IEEE Trans. on Computers*, Vol. C-28, pp831-834.

[5] Yun, D.Y.Y. (1978). Equivalent Complexity of a Class of Algebraic Mappings, *Proc. SIAM 1978 Spring Meeting*, Madison, Wisconsin.

[6] Taylor, F.J. and A. S. Ramnarayanan. (1981). An Efficient Residue-to-decimal Converter, *IEEE Trans. Circuits Syst.*, Vol. CAS-28, pp1164-1169.

[7] Vu, T.V. (1985). Efficient Implementations of the Chinese Remainder Theorem for Sign Detection and Residue Decoding, *IEEE Trans. Computers*, Vol C-34, pp646-651.

[8] Zhang, C.N., B. Shirazi and D.Y.Y. Yun. (1992). Efficient Algorithms for Residue Number Conversions, *Tech. report, Dept. of Computer Science, University of Regina, Sep. 1992.*